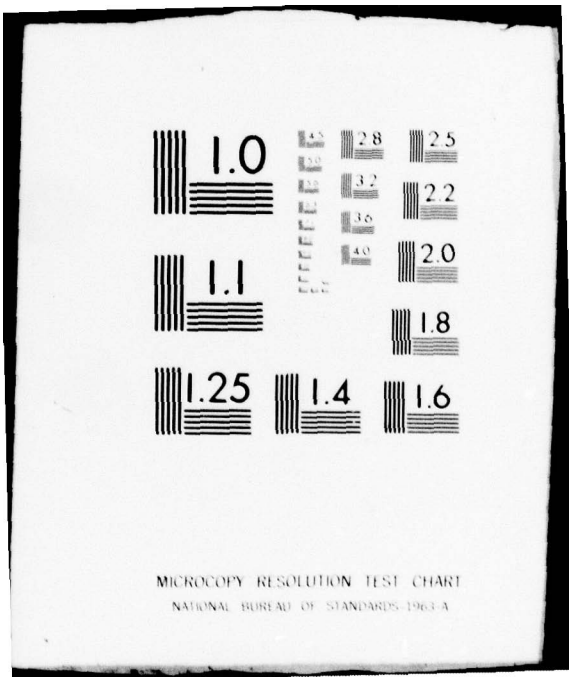
AD-A073 376 ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CEN--ETC F/G 9/2
THE IMS DATA STORAGE HIERARCHY - DSH-11.(U)
AUG 79 C LAM, S E MADNICK
UNCLASSIFIED CISR-M010-7908-03 N00039-78-G-0160
NL

| OF |
AD A073376

END
DATE FILMED
10-79
DDC


1.0
1.1
1.25
1.4
1.6
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

LEVEL II







Center for Information Systems Research
Massachusetts Institute of Technology
Alfred P. Sloan School of Management
50 Memorial Drive
Cambridge, Massachusetts, 02139
617 253-1000

Basic Ordering Agreement No. N00039-78-G-0160

Task No. 003

Internal Report No. M010-7908-03





Technical Report No. 3

# The IMS Data Storage Hierarchy - DSH-11

Chat-Yu Lam

Stuart E. Madnick

August 1979



Principal Investigator:

Professor Stuart E. Madnick

Prepared for:

Naval Electronics Systems Command

Washington, D.C.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER (9) Technical Report No. 3 | 2. GOVT ACCESSION NO. |
| 3. RECIPIENT'S CATALOG NUMBER | |
| 4. TITLE (and Subtitle) (6) The IMS Data Storage Hierarchy - DSH-11 | 5. TYPE OF REPORT & PERIOD COVERED TR# |
| 6. PERFORMING ORG. REPORT NUMBER MOIO-7908-03 | |
| 7. AUTHOR(s) (10) Chat-Yu/Lam, Stuart E./Madnick | 8. CONTRACT OR GRANT NUMBER(s) (15) N00039-78-G-0160-001 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Information Systems Research M.I.T. Sloan School of Management, Cambridge, MA 02139 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 53p. |
| 11. CONTROLLING OFFICE NAME AND ADDRESS (11) | 12. REPORT DATE August 1979 |
| | 13. NUMBER OF PAGES 52 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | 15. SECURITY CLASS. (of this report) UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

approved for public release; distribution unlimited

(14) CISR-MOIO-7908-03, CISR-TR-3

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

database management, database computer, multiprocessor system, hierarchical system, storage hierarchy, automatic data repair

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

409 590 JCR

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract

The need for efficient storage and processing of very large data bases coupled with advances in Large Scale Integration (LSI) technology have motivated various proposals to develop specialized computers for data base processing. The IMS data base computer (DBC) approach to this problem is to implement the logical functions of a data base management system by means of a hierarchy of microprocessors and to use a data storage hierarchy for efficient storage and retrieval of very large databases.

In a previous report, DSH-1, a general structure of the data storage hierarchy of the IMS DBC was described. DSH-1 incorporates novel features to enhance its availability and performance. Many alternative architectures of data storage hierarchies can be derived from DSH-1. These structures can be used to perform detail studies of various research issues concerning data storage hierarchies.

This report describes a simple structure of the IMS data storage hierarchy derived from DSH-1. This structure is called DSH-11 and is used to develop detail data movement algorithms for supporting the read and write operations. Multi-level inclusion properties of DSH-11 are then discussed.

## Preface

The Center for Information Systems Research (CISR) is a research center of the M.I.T. Sloan School of Management. It consists of a group of management information systems specialists including : faculty members, full-time research staff, and student research assistants. The Center's general research thrust is to devise better means for designing, implementing, and maintaining application software, information systems, and decision support systems.

Within the context of the research effort sponsored by the Naval Electronics Systems Command under contract N00039-78-G-0160, CISR has proposed to conduct basic research on the Intelligent Memory System (IMS). The IMS is a high performance, high availability information management system for supporting future Command, Communication and Control Systems.

Current advances in LSI and Multi-Chip Integration technology offer the potential for development of modular multi-processor building blocks for information management, as well as for intelligent memory controllers. Advances in information management technologies have made it possible to hierarchically organize the information management functions so as to facilitate pipeline and parallel processing. The IMS attempts to integrate the above hardware and software advances. In the IMS, all the information management functions are decomposed into a functional

hierarchy. Each level of the functional hierarchy is implemented using modular multi-processor building blocks. An automatic storage hierarchy is used by the IMS for storage and retrieval of very large databases. Each level of the storage hierarchy is implemented using modular multi-processor controllers and their associated storage devices.

The proposed research described in Contract N00039-78-G-0160 focuses on the concept development, architectural design and evaluation of the IMS storage hierarchy. Specific research tasks to be accomplished are : (1) design of a general structure of the IMS storage hierarchy, (2) design of a revised structure of the IMS storage hierarchy, (3) develop algorithms for the IMS storage hierarchy, (4) performance evaluation of the IMS storage hierarchy.

Technical Report No. 1 introduces the concepts of IMS and its research directions. Technical Report No. 2 discusses the concepts of data storage hierarchies from a practical point of view. A design of DSH-1, the data storage hierarchy of the IMS database computer, is described. This report describes a simple structure of the IMS data storage hierarchy derived from DSH-1. Algorithms for supporting the read and write operations are developed.

## ABSTRACT

The need for efficient storage and processing of very large data bases coupled with advances in Large Scale Integration (LSI) technology have motivated various proposals to develop specialized computers for data base processing. The IMS data base computer (DBC) approach to this problem is to implement the logical functions of a data base management system by means of a hierarchy of microprocessors and to use a data storage hierarchy for efficient storage and retrieval of very large databases.

In a previous report, DSH-1, a general structure of the data storage hierarchy of the IMS DBC was described. DSH-1 incorporates novel features to enhance its availability and performance. Many alternative architectures of data storage hierarchies can be derived from DSH-1. These structures can be used to perform detail studies of various research issues concerning data storage hierarchies.

This report describes a simple structure of the IMS data storage hierarchy derived from DSH-1. This structure is called DSH-11 and is used to develop detail data movement algorithms for supporting the read and write operations.

Multi-level inclusion properties of DSH-11 are then discussed.

TABLE OF CONTENTS

## Section I

## INTRODUCTION

To support better decision-making, effective and efficient storage and processing of very large data bases has been a major concern of modern organizations. Data Base Management Systems (DBMS's) have been developed specially to handle the storage, retrieval, and update of very large databases. Current DBMS's are capable of handling large databases in the order of a trillion bits of data, and capable of handling query rates of up to one hundred queries per second. Due to the increasing need for better information, and the declining costs of processors and storage devices, it is expected that future high performance DBMS's will be required to handle query rates and provide storage capacities several orders of magnitude higher than today's (Madnick, 1977). Furthermore, with such high query rates (generated by terminal users as well as directly from other computers), it is essential that the DBMS maintains non-stop operation. Thus, guaranteeing the availability of the DBMS becomes very important.

To meet these orders of magnitude improvements in performance over current DBMS's while at the same time providing high availability, new software and hardware architectures are needed. Various approaches have been proposed to develop high performance computers specially designed for database handling. These various approaches are discussed in (Lam and Madnick, 1979a; Lam and Madnick, 1979c). The approach taken by the IMS Data Base Computer is to make use of a hierarchy of microprocessors to implement the logical functions of a DBMS. A data storage hierarchy is used for storage and retrieval of large databases.

A data storage hierarchy is a storage subsystem with storage devices of different cost/performance characteristics arranged in a hierarchy, designed specifically for very large data bases. A data storage hierarchy takes advantage of the combination of different storage devices and locality of references (Denning, 1970) to realize a storage subsystem with very large capacity and small access time. Advances in memory technology have produced a more continuous, but wide spectrum of storage devices with different cost/performance characteristics. It is expected that this situation will persist. Thus, there is great potential in the development of effective data storage hierarchies.

A general structure of the IMS data storage hierarchy has been developed (Lam and Madnick, 1979d). This report is focused on developing effective data movement algorithms for supporting reading and writing of data in the IMS data storage hierarchy. The following sections will describe a simple structure of the IMS data storage hierarchy called DSH-11. Detail algorithms for supporting the read and write operations are developed. Multi-level inclusion properties of DSH-11 are then discussed.

## Section II

## STRUCTURE OF DSH-11

The general structure of DSH-11 is illustrated in Figure 2.1. There are h storage levels in DSH-11, denoted by L(1), L(2), ... , L(h). L(1) is the highest performance storage level. L(1) consists of k memory ports. Each memory port is connected to a processor. All k memory ports share the same local bus. A storage level controller (SLC) couples the local bus to the global bus. The global bus connects all the storage levels.

All other storage levels have the same basic structure. In each storage level, there is an SLC which couples the local bus to the global bus. There is a memory request processor (MRP) that handles requests to the storage level. There are a number of storage device modules (SDM's) that store the data within the storage level. The SLC, MRP, and SDM's share the same local bus.

The number of SDM's in different storage levels may be different. The type of storage device in the SDM's in different storage levels are different. For high efficiency, the block sizes of different storage levels are different.

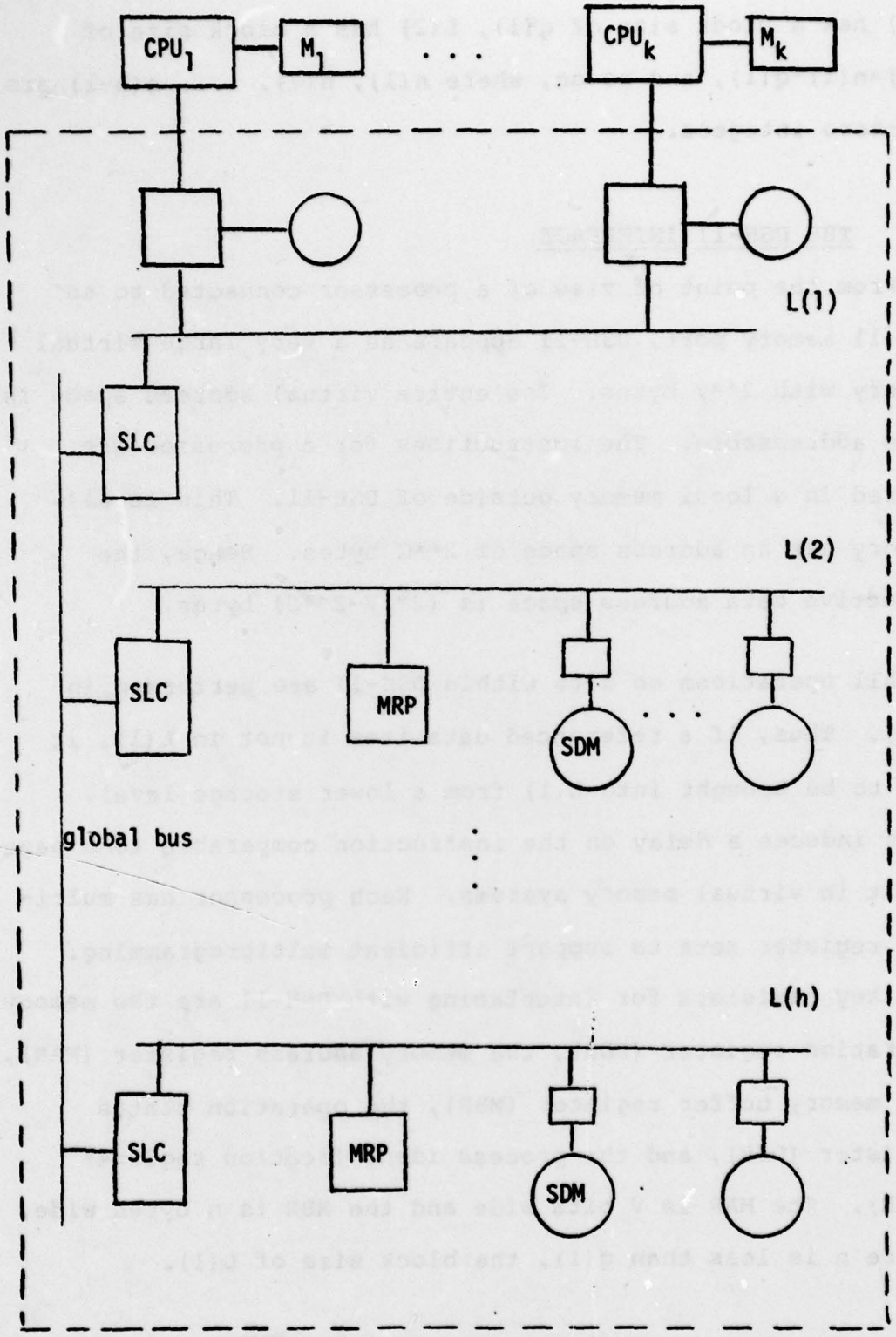



Figure 2.1 Structure of DSH-11

L(1) has a block size of q(1), L(2) has a block size of q(2)=n(1)*q(1), and so on, where n(1), n(2), ..., n(h-1) are non-zero integers.

## 2.1 THE DSH-11 INTERFACE

From the point of view of a processor connected to an DSH-11 memory port, DSH-11 appears as a very large virtual memory with 2**V bytes. The entire virtual address space is byte addressable. The instructions for a processor are stored in a local memory outside of DSH-11. This local memory has an address space of 2**G bytes. Hence, the effective data address space is (2**V-2**G) bytes.

All operations on data within DSH-11 are performed in L(1). Thus, if a referenced data item is not in L(1), it has to be brought into L(1) from a lower storage level. This induces a delay on the instruction comparable to a page fault in virtual memory systems. Each processor has multiple register sets to support efficient multiprogramming. The key registers for interfacing with DSH-11 are the memory operation register (MOR), the memory address register (MAR), the memory buffer register (MBR), the operation status register (OSR), and the process identification register (PIR). The MAR is V bits wide and the MBR is n bytes wide, where n is less than q(1), the block size of L(1).

A read operation requests n bytes of data at location pointed to by MAR to be brought into the MBR. A write operation requests the n bytes of data in the MBR be written to the location pointed to by the MAR. We shall assume that the n bytes of data in a memory reference do not cross a L(1) block boundary (If a data item crosses block boundaries, multiple requests will be used so that each request only reference data within block boundaries).

A read or write operation may proceed at the speed of the L(1) devices when the referenced data is found in L(1). Otherwise, the operation is interrupted and the processor switches to another process while DSH-11 moves the referenced data into L(1) from a lower storage level. When the data is copied into L(1), the processor is again interrupted to complete the operation.

## 2.2 THE HIGHEST PERFORMANCE STORAGE LEVEL - L(1)

As illustrated in Figure 2.1, L(1) consists of k memory ports. Each memory port consists of a data cache controller (DCC) and a data cache duplex (DCD). A DCC communicates with the processor connecting to the memory port. A DCC also maintains a directory of all data in the DCD. All k memory ports share a local bus. An SLC serves as a gateway for communciation between L(1) and other storage levels.

## 2.3 A TYPICAL STORAGE LEVEL - L(I)

A typical storage level, L(1), consists of a number of SDM's, an MRP, and an SLC. An SLC serves as the gateway for communication among storage levels. The MRP services requests to L(i). An SDM performs the actual reading and writing of data. An SDM consists of a device controller and the actual storage device.

To gain high throughput, communications over the global bus is in standard size packets. The packet size is such that it is sufficient for sending one L(1) data block. Communciations over a local bus at L(i) is also in standard size packets. The size of a packet depends on the storage level and is chosen so that a packet is sufficient to send a data block of size q(i).

In the following sections, the read and write operations are discussed in detail.

## Section III

## ALGORITHMS FOR SUPPORTING THE READ OPERATION

### 3.1 THE READ-THROUGH OPERATION

A read request is issued by a processor to its data cache controller. The data cache controller checks its directory to see if the requested data is in the data cache. If the data is found in the data cache, it is retrieved and returned to the processor. If the data is not in the data cache, a read-through request is queued to be sent to the next lower storage level, L(2), via the storage level controller.

At a storage level, a read-through request is handled by the memory request processor. The memory request processor checks its directory to determine if the requested data is in one of the storage device modules at that level. If the data is not in the storage level, the read-through request is queued to be sent to the next lower storage level via the storage level controller.

If the data is found in a storage level, L(i), a block containing the data is retrieved by the appropriate storage device module and passed to the storage level controller.

The storage level controller broadcasts the block to all upper storage levels in several standard size packets. Each upper storage level has a buffer to receive these packets. A storage level only collect those packets that assemble into a sub-block of the appropriate size that contains the requested data. This sub-block is then stored in a storage device module.

At L(1), the sub-block containing the requested data is stored, and the requested data is sent to the processor with the proper identification.

Figure 3.1 illustrates the read-through operation. Assume that DSH-11 has only three storage levels, L(1) with block size b, L(2) with block size 2b, and L(3) with block size 4b. Suppose a reference to a data item 'x' is found in a block in L(3). Then the sub-block of size 2b containing 'x' is broadcasted to L(2) and L(1) simultaneously. L(2) will accept and store the entire sub-block of size 2b. L(1) will only accept and store a block of size b that contains 'x'. The two sub-blocks, each of size 2b of a parent block in L(3) are refered to as the child blocks of the parent block.

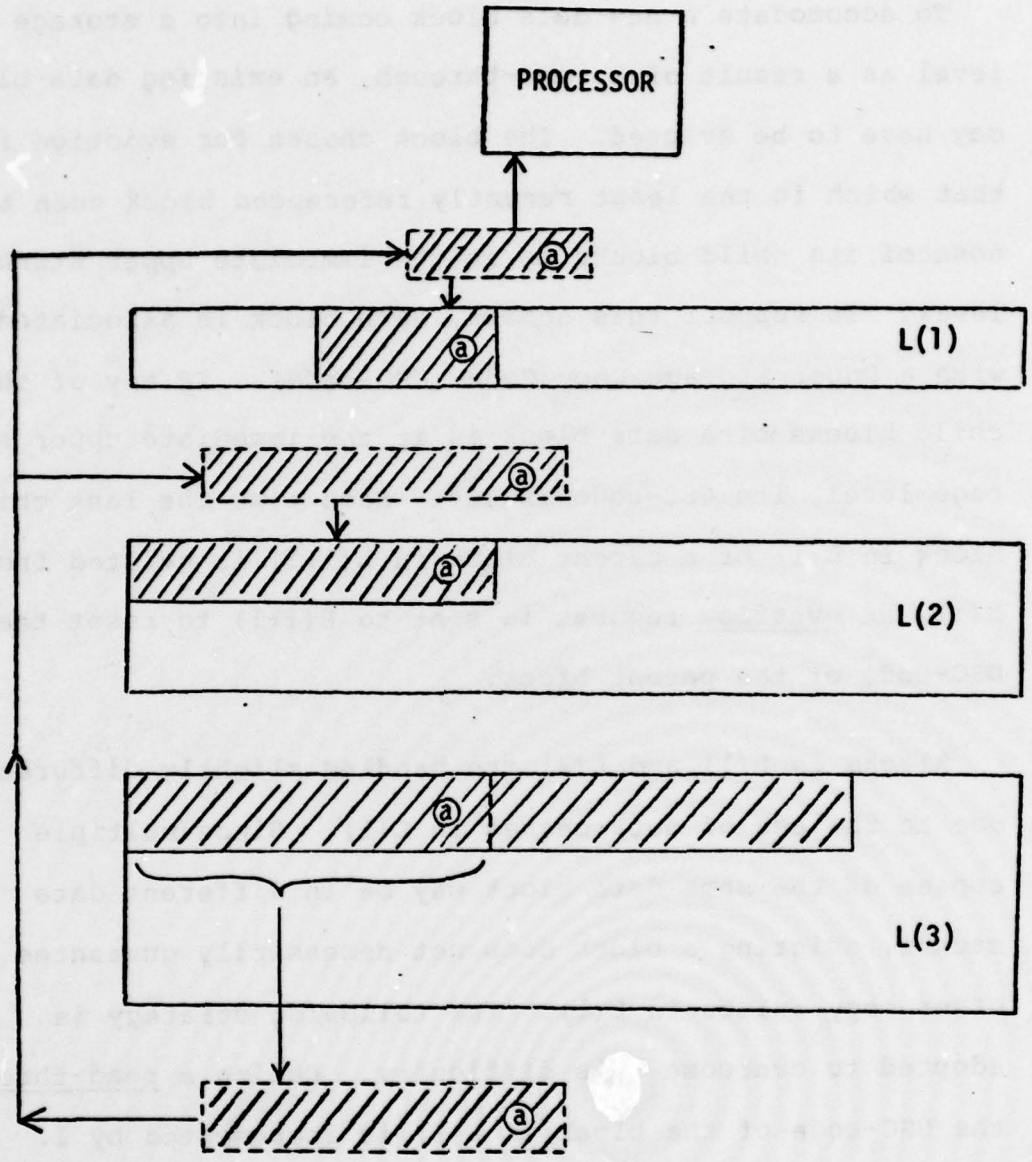



Figure 3.1 Read-Through Operation

## 3.2 OVERFLOW HANDLING

To accomodate a new data block coming into a storage level as a result of a read-through, an existing data block may have to be evicted. The block chosen for eviction is that which is the least recently referenced block such that none of its child blocks is in the immediate upper storage level. To support this scheme, each block is associated with a Upper Storage Copy Code (USC-code). If any of the child blocks of a data block is in the immediate upper storage level, its USC-code is set. Each time the last child block in L(i) of a parent block in L(i+1) is evicted from L(i), an overflow request is sent to L(i+1) to reset the USC-code of the parent block.

Blocks in L(1) and L(2) are handled slightly differently due to the use of data caches in L(1). Since multiple copies of the same data block may be in different data caches, evicting a block does not necessarily guarantee no other copy exists in L(1). The following strategy is adopted to overcome this difficulty. During a read-through, the USC-code of the block in L(2) is incremented by 1. Each time a block in L(1) is evicted, an overflow request is sent to L(2) to decrement the USC-code of the corresponding parent block. This strategy does not require communications among different data caches.

## 3.3 PATHOLOGICAL CASES OF READ-THROUGH

The parallel and asynchronous operations of DSH-11 and the use of buffers at each storage level complicates algorithms for handling the read operation. Pathological cases that affect the algorithms are discussed below.

### 3.3.1 Racing Requests

Two different requests R1 and R2 may reference the same block of data. Furthermore, these two requests may be close to each other such that both may be reading-through the same block of data at some storage level. Since a data block is transmitted in several packets asynchronously, each packet must be appropriately identified to avoid confusion when assembling the data sub-blocks at higher storage levels.

A similar situation arises when R1 and R2 are close together such that R2 begins to read-through the same data block that has just been read-through by R1. Thus a data block arriving at a storage level may find that a copy of it already exists at that storage level. In this case, the incoming data block is ignored. At L(1), this data block is ignored and the one in the data cache is read and returned to the processor, since this is the most up-to-date copy of the data block.

### 3.3.2 Erronous Overflow

When a data block is evicted from L(i) to make room for an incoming data block, an overflow request containing the virtual address of the evicted data block may be generated. The purpose of the overflow request is to inform L(i+1) that there is no longer any data block in L(i) with the same family address as the virtual address in the overflow request. Hence, an overflow request is generated only when the last member of a family in L(i) is evicted.

The overflow request has to be forwarded to the MRP at L(i+1). At any point on the way to the MRP, a data block in the same family as the evicted block may be read-through into L(i). This poses the danger that when the MRP receives the overflow request indicating that no data block in the same family as the evicted block exists in L(i), there is actually one such block being placed in L(i).

The following strategy is incorporated in the algorithms that support the read-through operation to avoid such a potential hazard.

1. At the time the overflow request is to be created in L(i), a check is made to see if there is any data block in the same family as the evicted block that is currently in any buffer of L(i) waiting to be placed in L(i). If so, the overflow request is not created.

2. At the time a new data block arrives in L(i), any overflow request with the same family address as

the incoming data block waiting to be sent to L(i+1) is purged.

3. When an overflow request arrives at L(i+1) from L(i), a check is made to see if there is any data block waiting to be sent to L(i) that has the same family address as the overflow request. If so, the overflow request is purged.

4. At the time a request is generated to send a data block to L(i), any overflow request from L(i) that is still waiting in L(i+1) that has the same family address as the data block to be sent to L(i) is purged.

### 3.3.3 Overflow to a Partially-assembled Block

Suppose that as a result of a read-through from L(i+2), B(i), the only child block of B(i+1), is in L(i) and B(i+1) is partly assembled in the buffer in L(i+1). It is possible that B(i+1) is still only partly assembled in L(i+1) when B(i) is evicted from L(i). The overflow request will find that the corresponding parent data block is still being assembled in L(i+1). A solution to this difficulty is to check, at the time of arrival of the overflow request, if there is any incoming data block which is the target parent block of the evicted block as indicated in the overflow request. If so, the overflow request is held till this parent block has been placed in a storage device.

## 3.4 TRANSACTIONS TO HANDLE THE READ OPERATION

A read request is issued by a processor to its data cache controller. If the data is not found in the data cache, it has to be brought up via a read-through. The read-through operation is realized via a number of transactions. The flow of transactions to support the read-through operation is illustrated in Figure 3.2.

A read-through transaction is created by a data cache controller and propagated to lower storage levels. At each storage level, the read-through transaction is handled by a memory request processor which checks its directory to see if the requested data is in the current storage level. If the data is not in the current storage level, the read-through transaction is sent to the next lower storage level.

Suppose that the data requested is found at L(i). The read-through transaction is terminated and a retrieve transaction is created to read the data from a storage device. A read-response-out transaction that contains the read data is sent to the storage level controller. The storage level controller generates a number of read-response-packet transactions which are broadcast to all higher storage levels. Each of these transactions contains a sub-block of the requested data.

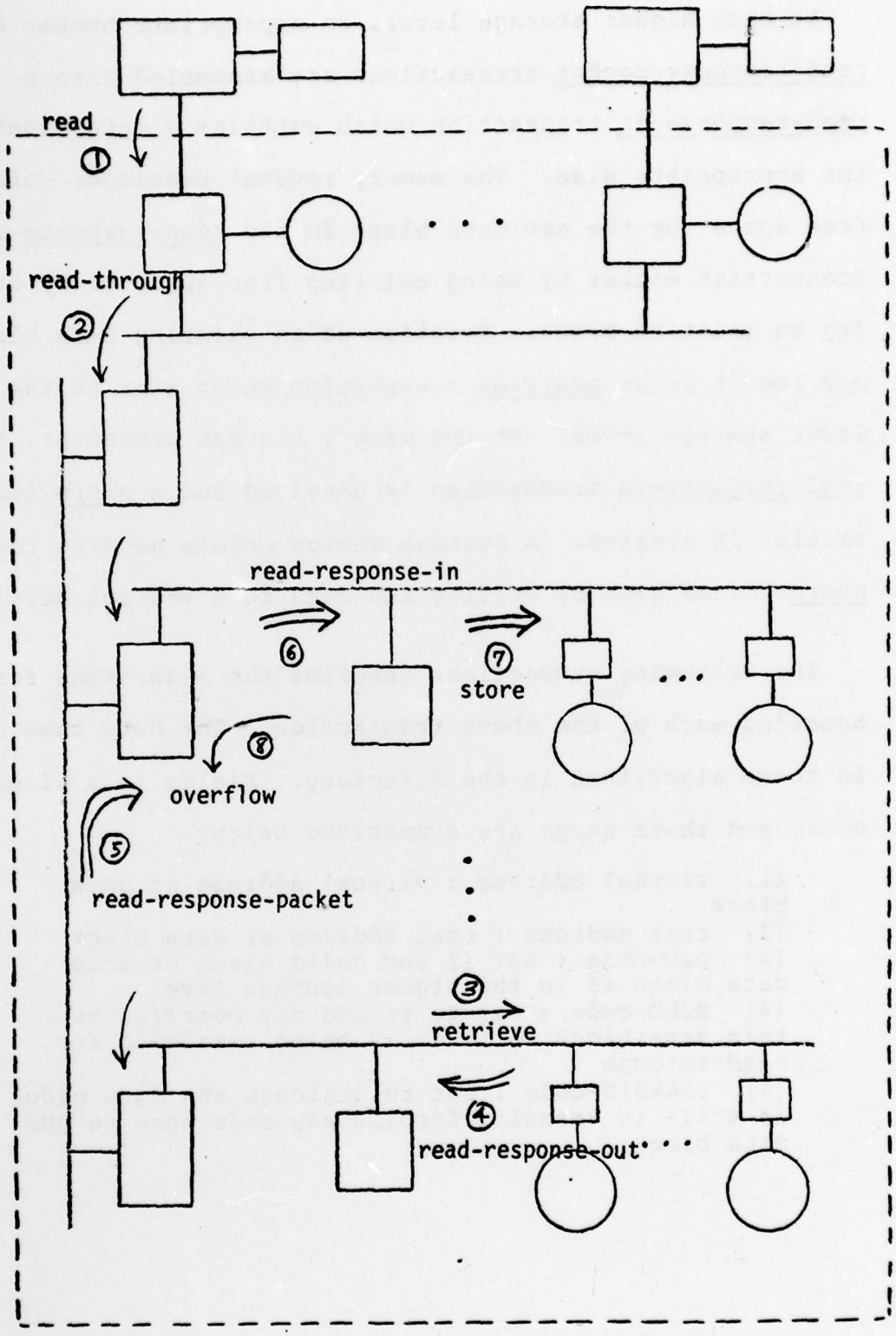



Figure 3.2 Transaction flow for read-through

At each higher storage level, an appropriate number of read-response-packet transactions are assembled into a read-response-in transaction which contains a data block of the appropriate size. The memory request processor obtains free space for the new data block in the read-response-in transaction either by using existing free space or by evicting an existing block. Eviction of an existing data block may result in an overflow transaction being sent to the next lower storage level. At the memory request processor, the read-response-in transaction is serviced and a store transaction is created. A storage device module handles the store transaction by writing the data to a storage device.

The following subsections describe the algorithms for handling each of the above transaction. The data base used in these algorithms is the directory. Fields in a directory entry and their usage are summarized below:

(1) virtual address : virtual address of data block
(2) real address : real address of data block
(3) USC-code : set if any child block of this data block is in the higher storage level
(4) HOLD-code : set to forbid any overflow to this data block while it is being retrieved for read-through
(5) TRANSIT-code : set to indicate the data block is still in transit, forbids any reference to the data block

### 3.4.1 The read-through Transaction

The read-through transaction is created by a data cache controller and propagated down the storage levels via the storage level controllers. It has the following format:

( read-through, virtual-address, process-id),

where virtual-address (denoted as read-through.virtual-addres) is the virtual address of the referenced data item, and process-id consists of a CPU identifier and a process number. It is the identifier of the process that generated the read operation. The transaction is handled by a memory request processor using the following algorithm.

1. Search directory for read-through.virtual-address.
2. If not found, forward the transaction to the storage level controller, which will send it to the next lower storage level.
3. If found, suppose it is in the i-th directory entry, and suppose directory(i).TRANSIT-code is not set, do:

   i) Set directory(i).USC-code to indicate a child block exists in the higher storage level for this block. If this is level L(2), increment directory(i).USC-code instead of setting it.

   ii) Set directory(i).HOLD-code to forbid any overflow to this block while the data is being retrieved.

   iii) Create a retrieve transaction :( retrieve, virtual-address,directory(i).real-address,process-id).

   iv) Send the retrieve transaction to the appropriate storage device module.

4. If found, suppose it is in the i-th directory entry, and suppose directory(i).TRANSIT-code is set, then hold the request and retry later. When the TRANSIT-code is set, it indicates that the corresponding data block is in transit, hence any reference to it is not allowed.

5. End.

### 3.4.2 The retrieve Transaction

The retrieve transaction is created by a memory request processor and handled by a storage device module as follows.

1. Read the data block using retrieve.real-address.

2. Create a read-response-out transaction :( read-response-out, virtual-address, process-id, data), where data is the block containing the referenced data item.

3. Send the read-response-out transaction to the storage level controller.

4. End.

### 3.4.3 The read-response-out Transaction

The read-response-out transaction is created by a storage device module and handled by a storage level controller using the following algorithm.

1. Purge any incoming overflow transaction that has the same family address as read-response-out.virtual-address.

2. Send ( update-directory, virtual-address,HOLD-code=0) to memory request processor to reset the HOLD-code, so that overflow to this block is now allowed.

3. Broadcast the transaction ( reserve-space, virtual-address, process-id) to all higher storage levels to reserve buffer space for assembling read-response-out.data.

4. Wait till all higer levels have acknowleged the space reservation transaction.

5. Generate the appropriate number of ( read-response-packet, virtual-address, process-id, data, data-virtual-address) transactions. Data is a standard size sub-block and data-virtual-address is the virtual address of this sub-block.

6. Broadcast each read-response-packet transaction to all higher storage levels.

7. End.

### 3.4.4 The read-response-packet Transaction

This transaction is created by a storage level controller and broadcast to all higher storage level controllers where they are assembled into read-response-in transactions to be handled by the memory request processors. Note that a storage level only accepts those packets relevant for assembling into a data block of the appropriate size, all other associated packets are ignored. The following algorithm is used by a storage level controller in assembling the read-response-in transactions.

1. If this is the first packet of the assembly, do:

   i) Purge any outgoing overflow transaction that has the same family address as the block being assembled.

   ii) Add the packet to the assembly.

2. If this is an intermediary packet of the assembly, simply add it to the assembly.

3. If this is the last packet of the assembly, do:

   i) Replace the assembly by a ( read-response-in, virtual-address, process-id, data) transaction. Data is the block just assembled.

   ii) Send the above transaction to the memory request processor.

4. End.

### 3.4.5 The read-response-in Transaction

This transaction is created by a storage level controller and sent to a data cache controller (for L(1)) or to a memory request processor (for L(2), L(3), ...).

The following algorithm is used by a data cache controller in handling this transaction.

1. Purge any outgoing overflow transaction that has the same family address as the block in the read-response-in transaction.

2. Search directory for read-response-in.virtual-address.

3. If found, suppose it is the i-th directory entry, do:

   i) Read data from the data cache using directory(i).real-address.

   ii) Send data to the processor.

   iii) Increment directory(i).USC-code by 1.

4. If not found, do:

i) Select a block to be evicted (assuming that data cache is full). This is the least recently referenced block such that it is not engaged in a stored-behind process. Suppose this block corresponds to directory(i).

ii) Obtain directory(i).virtual-address, directory(i).USC-code, and directory(i).real-address.

iii) Write read-response-in.data into location directory(i).real-address in the data cache.

iv) Return read-response-in.data to the processor.

v) Create ( overflow, directory(i).virtual-address, USC-code=directory(i).USC-code) transaction, send it to the storage level controller.

vi) Update directory(i).virtual-address with read-response-in.virtual-address.

vii) Set directory(i).USC-code to 1.

5. End.

At a memory request processor, the read-response-in transaction is handled as follows.

1. Purge any outgoing overflow transaction with the same family address as the data block in the read-response-in transaction.

2. Search for read-response-in.virtual-address in the directory.

3. If not found, do:

i) Select a block to be evicted (assuming that the storage level is full). This is the least recently referenced block such that it is not engaged in a store-behind process, it is not held (i.e., HOLD-code = 0), and it is not in transit (i.e., TRANSIT-code = 0). Suppose this block corresponds to directory(i).

ii) Obtain directory(i).virtual-address and directory(i).real-address.

iii) If the evicted block is the last of its family in the storage level and that there is no incoming block with the same family address then create a ( overflow, directory(i).virtual-address, USC-code=1) transaction. Send the transaction to the storage level controller to be sent to the next lower storage level.

iv) Set directory(i).TRANSIT-code to 1 to indicate the corresponding block is in transit.

v) Update directory(i).virtual-address with read-response-in.virtual-address.

vi) Set directory(i).USC-code to 1.

vii) Create a ( store, directory(i).real-address, data) transaction and send it to the appropriate storage device module.

4. End.

### 3.4.6 The store Transaction

This transaction is handled by a SDM. Store.data is placed in store.location, and a transaction ( update-directory, virtual-address, TRANSIT-code = Ø) is sent to the MRP to reset the TRANSIT-code so that references to this block is now allowed.

### 3.4.7 The overflow Transaction

This transaction is created by a data cache controller or a memory request processor and routed to a memory request processor in the lower storage level via the storage level

controllers. At each stop on the way to a memory request processor, a check is made to see if any incoming data block has the same family address as the overflow transaction. If so, the following algorithm is executed.

1. If the direction of flow of the overflow and read-response-in are opposite, the overflow is purged.
2. If the direction of flow of the overflow and the read-response-out are opposite, the overflow is purged.
3. If the two transactions are in the same direction of flow, the overflow is held to be processed after the read-response-in is handled.

At a memory request processor, if the HOLD-code is set for the parent block of the overflowed block, the overflow transaction is purged (HOLD-code is set indicates that the block is being retrieved by an SCM to be read-through to all upper storage levels). Otherwise, the USC-code of the parent block is decremented by overflow.USC-code.

## Section IV

## ALGORITHMS TO SUPPORT THE WRITE OPERATION

Algorithms to support the write operation are simplified by the multi-level inclusion properties of DSH-11. The multi-level inclusion properties of DSH-11 guarantee that all the data items in L(i) is contained in L(i+1). Thus, when writing a child block in L(i) to its parent block in L(i+1), the parent block is guaranteed to exist in L(i+1). The multi-level inclusion properties of DSH-11 will be discussed in a later section.

### 4.1 THE STORE-BEHIND OPERATION

After a block is placed in a data cache as a result of a read-through operation, its parent block exists in L(2), and its grand-parent block exists in L(3), and so on. Due to the multi-level inclusion properties of DSH-11, this situation will persist as long as the block is in the data cache.

After a data block in a data cache is updated, it is sent down to the next lower storage levels to replace the corresponding child block in its parent block. This updated parent block is sent down to the next lower storage level to

update its parent block, and so on. This process is refered to as the store-behind operation and takes place at slack periods of system operation.

DSH-11 uses a two-level store-behind strategy. This strategy ensures that an updated block will not be considered for eviction from a storage level until its parent and grand-parent blocks are updated. This scheme will ensure that at least two copies of the updated data exists in DSH-11 at any time. To support this scheme, a Store-Behind Code (SB-code) is associated with each data block in a storage level. The SB-code indicates the number of acknowledgements from lower storage levels that the block must receive before it can be considered for eviction.

In a write operation, the data item is written into the data cache duplex, and the processor is notified of the completion of the write operation. we shall assume that the data item to be written is already in L(1) (This can be realized by reading the data item into L(1) before the write operation). A store-behind operation is next generated by the data cache controller and sent to the next lower storage level. The block in L(1) that has just been updated is marked with a count of 2. This is illustrated in Figure 4.1(a).

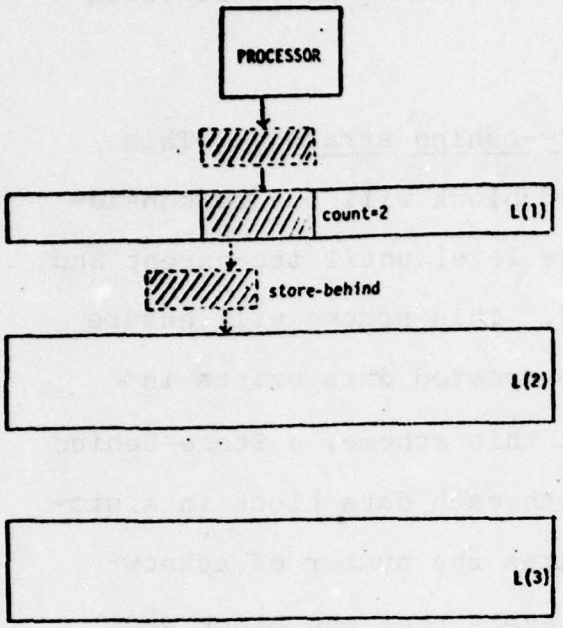



Figure 4.1(a) store-behind(a)

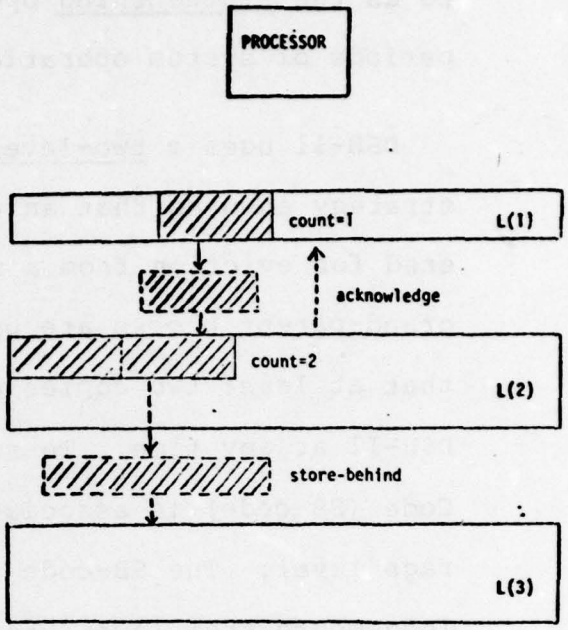



Figure 4.1(b) store-behind(b)

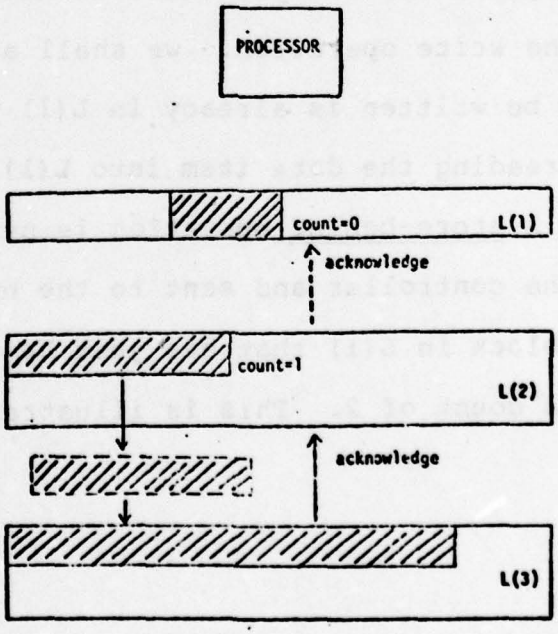



Figure 4.1(c) store-behind(c)

When a store-behind operation is received in L(2), the addressed data is written, and marked with a count of 2. An acknowledgement is sent to the next upper storage level, L(1), and a store-behind operation is sent to the next lower storage level, L(3). When an acknowledgement is received at L(1), the counter for the addressed data item is decremented by 1, which becomes 1. This is illustrated in Figure 4.1(b).

The store-behind is handled in L(3) by updating the appropriate data block. An acknowledgement is sent to L(2). At L(2), the corresponding block counter is decremented by 1, which becomes 1. The acknowledgement is forwarded to L(1). At L(1), the corresponding block counter is decremented by 1 which now becomes 0, hence the block is elligible for replacement. This is illustrated in Figure 4.1(c).

Thus we see that the two-level store-behind strategy maintains at least two copies of the written data at all times. Furthermore, lower storage levels are updated at slack periods of system operation, thus enhancing performance. Detail algorithms for supporting this scheme will be discussed in a later section.

## 4.2 LOST UPDATES

Several different updates to the same block will result in several different store-behind requests be sent to the next lower storage level. It is possible that these store-behind requests may arrive at the next storage level out of sequence, resulting in lost updates.

To resolve this potential hazard, there is a time-stamp associated with each block indicating the last time the block was updated. There is also a time-stamp associated with each child block of the parent block indicating the last time the child block was updated by a store-behind operation. A store-behind request will contain the block to be updated and its time-stamp. This time-stamp will be compared with that of the corresponding child block in the target parent block. Only when the store-behind data is more recent will the update to the target parent block be performed.

## 4.3 TRANSACTIONS TO SUPPORT THE WRITE OPERATION

Figure 4.2 illustrates the transactions to support the write operation. We shall assume that the target block of a write operation already exists in a data cache. This can be ensured by first reading the target block before issuing the write request to the data cache. After the data is written into a target data block in a data cache, a store-behind

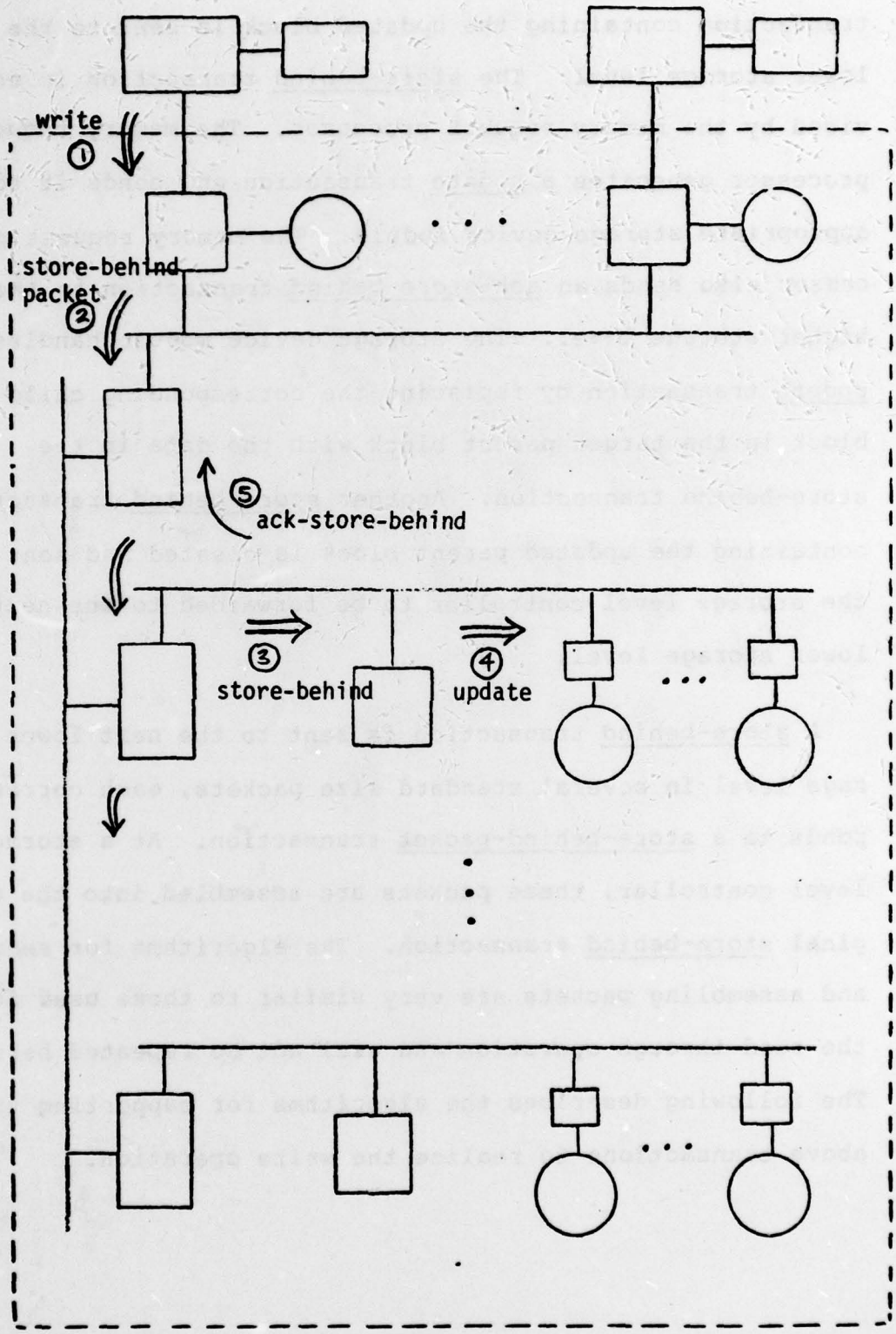



Figure 4.2 Transaction flow for store-behind

transaction containing the updated block is sent to the next lower storage level. The store-behind transaction is serviced by the memory request processor. The memory request processor generates a update transaction and sends it to the appropriate storage device module. The memory request processor also sends an ack-store-behind transaction to the higher storage level. The storage device module handles the update transaction by replacing the corresponding child block in the target parent block with the data in the store-behind transaction. Another store-behind transaction containing the updated parent block is created and sent to the storage level controller to be forwarded to the next lower storage level.

A store-behind transaction is sent to the next lower storage level in several standard size packets, each corresponds to a store-behind-packet transaction. At a storage level controller, these packets are assembled into the original store-behind transaction. The algorithms for sending and assembling packets are very similar to those used for the read-through operation and will not be repeated here. The following describes the algorithms for supporting the above transactions to realize the write operation.

### 4.3.1 The store-behind Transaction

A store-behind transaction has the following format:

( store-behind, virtual-address,process-id,data,time-stamp)

This transaction is handled by a memory request processor using the following algorithm.

1. Search directory for store-behind.virtual-address.
2. If not found, hold the transaction and retry after a time out, because the target parent block is still being assembled in the buffer.
3. If found, compare store-behind.time-stamp with the time-stamp of the corresponding child block of the target parent block.
4. If store-behind.data is more current than the child block, do:
   - i) Send ( update, virtual-address, data, real-address, time-stamp-of-parent) to the appropriate storage device module.
   - ii) Update the time-stamp of the child block with store-behind.time-stamp.
   - iii) Send ( ack-store-behind, virtual-address, process-id, ACK-code = 2) to the immediate higher storage level. ACK-code indicates the number of levels this transaction is to be routed upwards.
5. If store-behind.data is not more current than data in storage level, send two ( ack-store-behind, virtual-address, process-id, ACK-code = 2) to the immediate higher storage level.
6. End.

### 4.3.2 The update Transaction

The update transaction is handled by a storage device module using the following algorithm.

1. Replace the appropriate child block in the target parent block by update.data.
2. The updated target parent block is retrieved.
3. Send ( update-directory, virtual-address, SB-code = 2) to the memory request processor to increment SB-code of the target parent block by 2.
4. ( store-behind, virtual-address, process-id,target-parent-block, time-stamp = update.time-stamp-of-parent) is sent to the storage level controller to be sent to the next lower storge level.
5. End.

### 4.3.3 The ack-store-behind Transaction

This transaction is handled by a memory request processor. The algorithm used is as follows.

1. The SB-code of the corresponding block is decremented by 1.
2. The ack-store-behind.ACK-code is decremented by 1.
3. If ack-store-behind.ACK-code is greater than Ø the forward the ack-store-behind to the immediate upper storage level.
4. End.

## Section V

## MULTI-LEVEL INCLUSION PROPERTIES

As a result of the read-through operation, the block read-through into L(1) leaves its 'shadow' in every lower storage level that participated in the read-through operation. Is it true then, that a storage level, L(i), always contains every data block in L(i-1)? When this is true, multi-level inclusion (MLI) is said to hold.

It has been formally proved in (Lam and Madnick, 1979b) that certain algorithms incorporating the read-through strategy can guarantee MLI provided that the relative sizes of the storage levels be appropriately chosen. Furthermore, it is found that certain other algorithms can never guarantee MLI. This section explores the MLI properties of DSH-11. In the following sections, the importance of MLI is briefly reviewed, a model of DSH-11 is developed, and the MLI property of DSH-11 is analyzed informally.

### 5.1 IMPORTANCE OF MLI

The MLI properties have important implications for the performance and availability of DSH-11. First, since the

block size of L(i) is larger than that of L(i-1), L(i) can be viewed as an extension of the spatial-locality (Madnick, 1973) of L(i-1). Second, except for the lowest storage level, each data item has at least two copies in different storage levels. Hence, even the failure of an entire storage level will not result in data loss. Third, algorithms to support the write operation is simplified because of MLI because a store-behind operation always finds the target parent data block exists in a storage level.

## 5.2 A MODEL OF DSH-11

Figure 5.1 illustrates a model of DSH-11. DSH-11 has h storage levels, L(1), L(2), ... , L(h). L(1) consists of k data caches. Each data cache consists of a buffer B(1,i), and a storage, M(1,i). All the buffers of the data caches are collectively denoted as B(1), and all the storage of the data caches are collectively denoted as M(1). The size of B(1,i) is b(1,i) number of blocks of size q(1). The size of M(1,i) is m(1,i) number of blocks of size q(1). Hence L(1) has b(1) = b(1,1) + b(1,2) + ... + b(1,k) blocks of buffer space and m(1) = m(1,1) + m(1,2) + ... + m(1,k) blocks of storage space.

A buffer is for holding data blocks coming into or going out of the storage level. A data block may be partially

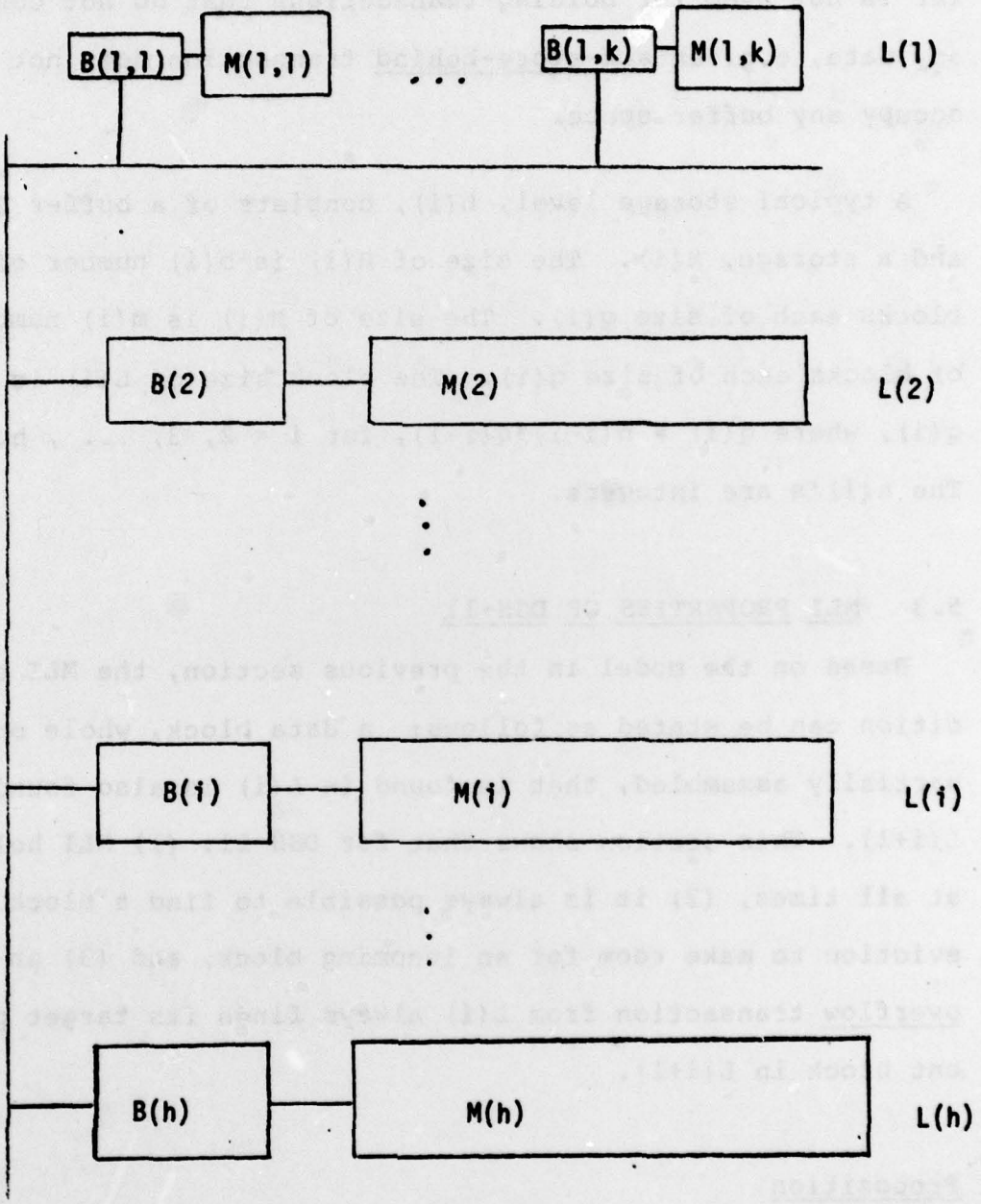



Figure 5.1 Model of DSH-11

assembled in a buffer. Only data blocks in the storage space are accounted for by the directory. Note that a buffer is not used for holding transactions that do not contain any data, e.g. an ack-store-behind transaction does not occupy any buffer space.

A typical storage level, L(i), consists of a buffer B(i), and a storage, M(i). The size of B(i) is b(i) number of blocks each of size q(i). The size of M(i) is m(i) number of blocks each of size q(i). The block size of L(i) is q(i), where q(i) = n(i-1)*q(i-1), for i = 2, 3, ... , h. The n(i)'s are integers.

## 5.3 MLI PROPERTIES OF DSH-11

Based on the model in the previous section, the MLI condition can be stated as follows: a data block, whole or partially assembled, that is found in L(i) is also found in L(i+1). This section shows that for DSH-11, (1) MLI holds at all times, (2) it is always possible to find a block for eviction to make room for an incoming block, and (3) an overflow transaction from L(i) always finds its target parent block in L(i+1).

**Proposition**

**Using the algorithms described in the previous sections, if m(i+1) is greater than m(i)+b(i) then**

1. MLI holds for L(i) and L(i+1), i.e., any block found in L(i) can be found in L(i+1),

2. If block replacement in M(i+1) is required, there is always a block not in L(i) that can be considered for overflow, and

3. An overflow transaction from L(i) always contains the address of a block that can be found in M(i+1).

Proof of Proposition

There are three cases:

1. There are no overflows from L(i): Since m(i+1) is greater than m(i)+b(i), no overflow from L(i) implies no overflow from L(i+1). Thus all blocks present in L(i) are still in L(i+1), i.e., (1) is true.

2. There are overflows from L(i), no overflow from L(i+1): No overflow from L(i+1) implies that all blocks referenced so far are still in L(i+1). Thus any block in L(i) is still in L(i+1), i.e., (1) is true. Since any overflow from L(i) will find the block still in L(i+1), (3) is true.

3. There are overflows from L(i+1): Consider the first overflow from L(i+1). Just before the overflow, (1) is true. Also just before the overflow, M(i+1) is full. M(i+1) is full and m(i+1) is greater than m(i)+b(i) implies that there is at least one block in M(i+1) that is not in L(i) (i.e., their USC-code = Ø). Choose from these blocks the least recently referenced block such that its SB-code = Ø. If no such block exists, wait, and retry later. Eventually the store-behind process for these blocks will be terminated and these blocks will be released. Thus a block will be available for overflow from M(i+1). Thus (2) is true. After the overflow, (1) is still preserved. (1) and (2) implies (3).

If next reference causes no overflow from L(i+1), then the arguement in Case 2 applies. If the next reference causes overflow from L(i+1), then the arguement in Case 3 applies.

## Section VI

## SUMMARY

The DSH-11 design, a data storage hierarchy for the IMS data base computer, is described. Algorithms for supporting the read and write operations in DSH-11 are described in detail. It is then shown that DSH-11 is able to guarantee multi-level inclusion at all times for any reference string provided that the sizes of the buffers and storage at the storage levels are chosen appropriately.

There are still many open research issues regarding the performance, theoretic properties and practical implementation of DSH-11. A key question to be addressed is how well DSH-11 will perform. Performance evaluation of DSH-11 will provide valuable insights for alternative structures and/or alternative data movement algorithms. To formally study the properties of DSH-11, formal descriptions of the algorithms have to be developed. The use of Petri Nets for this purpose may be appropriate due to the asynchronous and highly parallel nature of these algorithms. When such formalisms are developed, the MLI properties of DSH-11 can then be formally proven. We have treated the various components (MRP,

SLC, etc.) as black boxes. Each of these components is actually a multiprocessor complex. It is necessary to specify the characteristics of the processor to be used as well as the interconnection structure. These research issues are currently being investigated as part of the IMS project.

## REFERENCES

(Denning, 1970): Denning, P.J., 'Virtual Memory', ACM Computing Surveys, 2, 3 (September 1970), 153-190.

(Madnick, 1973): Madnick, S.E., 'Storage Hierarchy Systems', MIT Project MAC Report No. TR-105, 1973.

(Madnick, 1977): Madnick, S.E., 'Trends in Computers and Computing: The Information Ultility', Science, 195, 4283 (1977), 1191-1199.

(Lam and Madnick, 1979a): Lam, C.Y., and Madnick, S.E., 'INFOPLEX Data Base Computer Architecture - Concepts and Directions', MIT Sloan School Working Paper No. 1046-79, 1979.

(Lam and Madnick, 1979b): Lam, C.Y., and Madnick, S.E., 'Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data', MIT Sloan School Working Paper No. 1047-79, 1979.

(Lam and Madnick, 1979c): Lam, C.Y., and Madnick, S.E., 'The Intelligent Memory System Architecture - Research Directions', MIT Sloan School of Management Internal Report No. M010-7908-01, August 1979.

(Lam and Madnick, 1979d): Lam, C.Y., and Madnick, S.E., 'The IMS Data Storage Hierarchy - DSH-1', MIT Sloan School Internal Report No. M010-7908-02, August, 1979.